Practical Lock-Free Implementation of LL/SC Using Only Pointer-size CAS

Gao, Hui; Fu, Yan; Hesselink, Wim H.

Published in:
EPRINTS-BOOK-TITLE

Practical Lock-Free Implementation of LL/SC Using Only Pointer-size CAS

Hui Gao, Yan Fu
School of Computer Science and Engineering
University of Electronic Science and Technology of China
Chengdu, China
Email: huigao@uestc.edu.cn, fuyan@ustec.edu.cn

Wim H. Hesselink
Dept. of Mathematics and Computing Science
University of Groningen
Groningen, The Netherlands
Email: w.h.hesselink@rug.nl

***Abstract*—The significant benefit of lock (or wait)-freedom for real-time systems is that by avoiding locks the potentials for deadlock and priority inversion are avoided. The lock-free algorithms often require the use of special atomic processor instructions such as *CAS* (compare and swap) or *LL/SC* (load linked/store conditional). However, many machine architectures support either *CAS* or *LL/SC* with restricted semantics. In this paper, we present a Practical lock-free implementation of the ideal semantics of *LL/SC* using only pointer-size *CAS*. To ensure our implementation is not flawed, we used the higher-order interactive theorem prover *PVS* for mechanical support.**

I. Introduction

Over the past two decades the research community has developed a body of knowledge concerning "Lock-Free" and "Wait-Free" algorithms and data structures. In contrast to algorithms that protect access to shared data with locks, lock-free and wait-free algorithms are specially designed to allow multiple threads to read and write shared data concurrently without corrupting it. The significant benefit of lock (or wait)-freedom for real-time systems is that by avoiding locks the potentials for deadlock and priority inversion are avoided.

In computer science, *non-blocking* synchronization ensures that threads competing for a shared resource do not have their execution indefinitely postponed by mutual exclusion. A non-blocking algorithm is *lock-free* if there is guaranteed system-wide progress; *wait-free* if there is also guaranteed per-thread progress. More formally, an algorithm is called *lock-free* if every step taken achieves global progress; an algorithm is called *wait-free* if every operation has a bound on the number of steps it will take before completing.

A number of researchers[3], [5], [7], [9], [15] have proposed techniques for designing lock-free implementations. The lock-free algorithms often require the use of special atomic processor instructions such as *CAS* (compare and swap) or *LL/SC* (load linked/store conditional). However, Current mainstream architectures support either *CAS* or *LL/SC* with restricted semantics (but not both), which are susceptible to the *ABA* problem [14].

The ideal semantics of the atomic primitives *LL/SC* are inherently immune to that problem. However, for practical architectural reasons, no processor architecture supports the ideal semantics of *LL/SC*. Designing efficient algorithms to bridge the gap has been the subject of many researchers' interest. However, most of the research is focused on implementing only small LL/SC objects, whose value fits in a single machine [4], [8], [9], [11].

In this paper, using only pointer-size *CAS* we present a Practical lock-free implementation of the ideal semantics of *LL/SC* objects (whose value does not have to fit in a single machine word) without causing ABA problem. To ensure our implementation is not flawed, we used the higher-order interactive theorem prover *PVS* [6] for mechanical support. All invariants as well as the simulation relation have been completely verified with *PVS*.

II. Preliminary

The machine architecture that we have in mind is based on modern shared-memory multiprocessors that can access a common shared address space in a heap. There can be several processes running on a single processor. Variables in shared context are visible to all processes running in associated parallel. Variables in private context are hidden from other processes.

We assume a universal set $\mathcal{V}$ of typed variables, which is called the *vocabulary*. A state s is a type-consistent interpretation of $\mathcal{V}$, mapping variables $v \in \mathcal{V}$ to values $s[\![v]\!]$. We denote by Σ the set of all states. If $\mathcal{C}$ is a command, we denote by $\mathcal{C}_p$ the transition $\mathcal{C}$ executed by process p, and $s[\![\mathcal{C}_p]\!]t$ indicates that in state s process p can do a step $\mathcal{C}$ that establishes state t. When discussing the effect of a transition $\mathcal{C}_p$ from state s to state t on a variable v, we abbreviate $s[\![v]\!]$ to v and $t[\![v]\!]$ to v'. We use the abbreviation $Pres(V)$ for $\bigwedge_{v \in V}(v' = v)$ to denote that all variables in the set V are preserved by the transition.

A. The Semantics of Synchronization Primitives

Traditional multiprocessor architectures have included hardware support only for low level synchronization primitives such as *CAS* and *LL/SC*, while high level synchronization primitives such as locks, barriers, and condition variables have to be implemented in software.

CAS atomically compares the contents of a location with a value and, if they match, stores a new value at the location. The semantics of *CAS* is given by equivalent atomic statements

978-0-7695-3887-7/09/$26.00 ©2009 IEEE

below. We use angular brackets $\langle \ldots \rangle$ to indicate atomic execution of the enclosed specification command.

proc *CAS*(**ref** X : `Val`; **in** old, new : `Val`) : `Bool` =
$\langle$ **if** $X = old$ **then** $X := new$; **return** *true*
else return *false*; **fi** $\rangle$

LL and *SC* are a pair of instructions, closely related to the *CAS*, and together implement an atomic Read/Write cycle. Instruction *LL* first reads the content of a memory location, say X, and marks it as "reserved" (not "locked"). If no other processor changes the content of X in between, the subsequent *SC* operation of the same process succeeds and modifies the value stored; otherwise it fails. The semantics of LL and SC are given by equivalent atomic statements below, where *me* is the process identifier of the acting process.

proc *LL*(**in** X : `Val`) : `Val` =
$\langle$ $S.X := S.X \cup \{me\}$; **return** X; $\rangle$

proc *SC*(**ref** X : `Val`; **in** Y : `Val`) : `bool` =
$\langle$ **if** $me \in S.X$ **then**
$S.X := \emptyset$; $X := Y$; **return** *true*
else return *false*; **fi** $\rangle$

B. Refinement mappings

In practice, the specification of systems is concerned rather with externally visible behavior than computational feasibility. We assume that all levels of specifications under consideration have the same observable state space Σ_0, and are interpreted by their observation functions $\Pi : \Sigma \to \Sigma_0$. Every specification can be modeled as a four-tuple $(\Sigma, \Pi, \Theta, \mathcal{N})$ where $(\Sigma, \Theta, \mathcal{N})$ is the *transition system* [2].

A *refinement mapping* from a lower-level specification $\mathcal{S}_c = (\Sigma_c, \Pi_c, \Theta_c, \mathcal{N}_c)$ to a higher-level specification $\mathcal{S}_a = (\Sigma_a, \Pi_a, \Theta_a, \mathcal{N}_a)$, written $\phi : \mathcal{S}_c \sqsubseteq \mathcal{S}_a$, is a mapping $\phi : \Sigma_c \to \Sigma_a$ that satisfies:

1) ϕ preserves the externally visible state component: $\Pi_a \circ \phi = \Pi_c$.
2) ϕ is a *simulation*, denoted $\phi : \mathcal{S}_c \preccurlyeq \mathcal{S}_a$:
   ① ϕ takes initial states into initial states: $\Theta_c \Rightarrow \Theta_a \circ \phi$.
   ② $\mathcal{N}_c$ is mapped by ϕ into a transition (possibly stuttering) allowed by $\mathcal{N}_a$:
   $\mathcal{Q} \wedge \mathcal{N}_c \Rightarrow \mathcal{N}_a \circ \phi$, where $\mathcal{Q}$ is an invariant of $\mathcal{S}_c$.

Below we need to exploit the fact that the simulation only quantifies over all reachable states of the lower-level system, not all states. We therefore explicitly allow an invariant $\mathcal{Q}$ in condition 2 ②. The following theorem is stated in [1].

Theorem 1 *If there exists a refinement mapping from $\mathcal{S}_c$ to $\mathcal{S}_a$, then $\mathcal{S}_c$ implements $\mathcal{S}_a$.*

III. The Lock-free Implementation of *LL*/*SC*

Let us assume there are P (≥ 1) concurrently executing sequential processes. To distinguish private persistent variables of different processes, every persistent private variable name can be extended with the suffix "." + "*process identifier*". In

Constant
$P =$ `number of processes`;
$N =$ `number of shared variables`;
Shared variable
`Node`: **array** $[1 \ldots N]$ **of** `Val`;
`S`: **array** $[1 \ldots N]$ **of** `Set`;
Private variable
pc: $\{a_1, a_2\}$;
me: `ProcID`;
proc *LL*(**in** $x : 1 \ldots N$) : `Val` =
a_1: $\langle S[x] := S[x] \cup \{me\}$; **return** `Node`$[x]$; $\rangle$
proc *SC*(**in** $x : 1 \ldots N$; Y : `Val`) : `Bool` =
a_2: $\langle$**if** $me \in S[x]$ **then**
$S[x] := \emptyset$; $Node[x] := Y$; **return** *true*
else return *false*; **fi** $\rangle$
Initial conditions
Θ_a: $\forall p$: $1 \ldots P$: $pc.p = a_1 \vee pc.p = a_2$

Fig. 1. The Specification $\mathcal{S}_a$ of *LL*/*SC*

particular, $pc.q$ is the program location of process q, it ranges over all defined integer labels.

The specification $\mathcal{S}_a$ of *LL*/*SC* can then be given as shown in Fig. 1. In the specification, we model the `Node` as an array of the N shared variables in the heap under consideration, which can be of any type (e.g. `Val`). The indices of the `Node` are the addresses (or the pointers) to shared variables. We can thus simply regard the shared variable X (under consideration) as a synonym of an index of the `Node`, and its value is stored in `Node`$[x]$. As before, the action enclosed by angular brackets $\langle \ldots \rangle$ is defined as atomic statement.

We now turn our attention to the lock-free implementation using only pointer-size *CAS*, which is given by the algorithm shown in Fig. 2. This lock-free implementation is inspired by our previous work [12].

In the lock-free implementation, the shared variable `indir`$[x]$ acts as pointers to the shared node x under consideration(i.e., the shared variable), while `node`$[mp_p]$ is taken as a "private" node of process p though it is declared publicly: other processes can read it but cannot modify it.

We need to ensure all indices of shared nodes and "private" nodes (declared in a public way) are mutually different. The basic idea to do this is to employ array `prot` to count the number of processes that are using an index for accessing a node, in such a way that the consistency of a node can be checked by its index: suppose process p first reads the index of node x to local variable m (see line c_{10} and line c_{22}), then the consistency can be checked later by the predicate $m =$ `indir`$[x]$.

In $\mathcal{S}_c$, *LL*/*SC* are taken as pairs of instructions, that together implement atomic read/write cycle. In the implementation, we therefore increment and decrement the corresponding counter (in array `prot`) in the procedures of *LL* and *SC*, respectively. The increment of `prot`$[m]$ (see line c_{14}) is essential since it guarantees that the accessing node can not be taken as a private

Constant
$P =$ `number of processes`;
$N =$ `number of shared variables`;
$K = N + 2P$;
Shared variable
`Node`: **array** $[1 \dots K]$ **of** `Val`;
`indir`: **array** $[1 \dots N]$ **of** $1 \dots K$;
`prot`: **array** $[1 \dots K]$ **of** $0 \dots K$;
Private persistent variable
pc: $[c_{10} \dots c_{34}]$;
mp: $1 \dots K$;
proc *LL*(**in** $x : 1 \dots N$) : `Val` =
loop
c_{10}: $m :=$ `indir`$[x]$;
c_{12}: *mybuf* := `Node`$[m]$;
c_{14}: `prot[m]++`;
c_{16}: **if** $m =$ `indir`$[x]$ **then**
`return` *mybuf*;
else
c_{18}: `prot`$[m]--$;
fi;
end;
proc *SC*(**in** $x : 1 \dots N$; $Y :$ `Val`) : `Bool` =
c_{20}: `Node`[*mp*] := Y
loop
c_{22}: $m :=$ `indir`$[x]$;
c_{24}: **if** *CAS*(`indir`$[x]$, m, *mp*) **then**
c_{26}: `prot[m]--`;
c_{28}: **if** `prot`$[m] = 1$ **then**
mp := m;
else
c_{30}: `prot`$[m]--$;
repeat
choose *mp* **from** $1 \dots K$
c_{32}: **until** *CAS*(`prot`[*mp*], 0, 1)
fi;
`return` *true*;
else
c_{34}: `prot`$[m]--$;
`return` *false*;
fi;
end.
Initial conditions
Θ_c: $(\forall p : 1 \dots P : (pc.p = c_{10} \vee pc.p = c_{20})$
$\wedge$ $mybuf_p = N{+}p)$
$\wedge$ $(\forall i : 1 \dots N :$ `indir`$[i] = i)$
$\wedge$ $(\forall i : 1 \dots K :$ `prot`$[i] = (i \leq N{+}P \ ? \ 1 : 0))$

Fig. 2. The Lock-free implementation $\mathcal{S}_c$ of *LL*/*SC*

node. The decrement of `prot`$[m]$ (see lines c_{18}, c_{26}*and*c_{30}) is also important since otherwise the implementation will eventually run out of memory.

At line c_{24}, after *CAS* succeeds the "private" node with value Y serves as the shared node of x If some other process successfully updates a shared node (line c_{24}) while an active process p is copying the shared node to its "private" node (see line c_{12}, process p will restart the loop, since its private view of the node is not consistent anymore.

During the read/write cycle. Decrement of `prot`$[m]$ in line c_{26} is necessary since m does not refer a shared node when *CAS* in line c_{24} succeeds. When the check in line c_{28} finds that `prot`$[m]$ equals 1, it means that only this process is hanging on that index, and the process can thus immediately treat that node as its private node. Otherwise, the previous shared node can not be served as a private node immediately when some process is still hanging on that node. Otherwise, interference may occur when the new "private" node is redirected to be a shared node again. Before the process starts to find an unused index for its private node, it needs to release the reading access to the node (see line c_{30}). Right after the success of *CAS* at line c_{32}, the "private' node indexed by *mp* has been successfully chosen. When a new unused index, say *mp*, is chosen in line c_{32}, the process will set `prot`[*mp*] to 1 instead of 0. Therefore, no other process will regard that chosen "index" as an unused index and take that for its private use.

Guarded by `prot`, every "private" node for each process is now truly private since it does not even allow some other process to have a peep at its content. This means that the assignment of the "private" node is safe (see line c_20), and it only needs to be executed once at the beginning of the procedure *SC*.

In the implementation $\mathcal{S}_c$, we introduce a constant $K \geq N + 2P$ for the sizes of the arrays `Node` and `prot`. There is a trade-off between space and time that the user can choose: large K is faster when choose an unused index for a "private node" at line c_{32}, but requires more space.

IV. Correctness

In this section we will prove that the concrete system $\mathcal{S}_c$ implements the abstract system $\mathcal{S}_a$. Formally, like we did in [10], [14], we define

$\Sigma_a \triangleq$ (`Node`$[1 \dots N]$, `S`) $\times$ (*pc*, *me*, x, Y)^P
$\Sigma_c \triangleq$ (`Node`$[1 \dots K]$, `indir`$[1 \dots N]$,
`prot`$[1 \dots K]$) $\times$ (*pc*, x, Y, *mp*, m, *mybuf*)^P
$\Pi_a(\Sigma_a) \triangleq$ `Node`$[1 \dots N]$
$\Pi_c(\Sigma_c) \triangleq$ `node`[`indir`$[1 \dots N]$]
$\mathcal{N}_a \triangleq \mathcal{N}_{a_0} \vee \mathcal{N}_{a_1} \vee \mathcal{N}_{a_2}$
$\mathcal{N}_c \triangleq \bigvee_{10 \leq i \leq 34} \mathcal{N}_{c_i}$.

The transitions of the abstract system can be described: $\forall s, t : \Sigma_a$, $p : 1 \dots P$:

$s[\![(\mathcal{N}_{a_0})_p]\!]t \triangleq$ $s = t$ (to allow stuttering)
$s[\![(\mathcal{N}_{a_1})_p]\!]t \triangleq$ $pc.p = a_1 \ \wedge \ pc'.p = a_2$
$\wedge$ `S'`$[x.p] =$ (`S`$[x.p] \cup$ *me*)
$\wedge$ $Pres(\mathcal{V} - \{pc.p,$ `S`$[x.p]\})$
$s[\![(\mathcal{N}_{a_2})_p]\!]t \triangleq$ $pc.p = a_2 \ \wedge \ pc'.p = a_1$
$\wedge$ $((me \in$ `S`$[x.p] \ \wedge$ `S'`$[x.p] = \emptyset \ \wedge$ `Node'`$[x.p] = Y$
$\wedge$ $Pres(\mathcal{V} - \{pc.p,$ `Node`$[x.p],$ `S`$[x.p]\}))$
$\vee (me \notin$ `S`$[x.p] \ \wedge \ Pres(\mathcal{V} - \{pc.p\})))$

The transitions of the concrete system can be described in the same way. Here we only provide the description of concrete transitions c_{16}: $\forall s, t : \Sigma_c,\ p : 1 \ldots P$:

$$
\begin{array}{l}
s[\![(\mathcal{N}_{c_{16}})_p]\!]t \triangleq\ pc.p = c_{16} \\
\quad \wedge\ ((m.p = \mathtt{indir}[x.p]\ \wedge\ pc'.p = c_{20}) \\
\qquad \vee\ (m.p \neq \mathtt{indir}[x.p]\ \wedge\ pc'.p = c_{18})) \\
\quad \wedge\ Pres(\mathcal{V} - \{pc.p\})
\end{array}
$$

To prove that $\mathcal{S}_c$ implements $\mathcal{S}_a$, we define the state mapping ϕ: $\Sigma_c \rightarrow \Sigma_a$ by showing how each component of Σ_a is generated from components in Σ_c:

$$
\begin{array}{l}
\forall i\text{: } 1 \ldots N\text{: } \mathtt{Node}_a[i] = \mathtt{Node}_c[\mathtt{indir}_c[i]] \\
\forall i\text{: } 1 \ldots N\text{: } \mathtt{S}_a[i] = \{p\text{: } 1 \ldots P | pc_c.p \notin \{c_{10}; c_{20}; c_{22}\} \\
\qquad \wedge\ x_c.p = i\ \wedge\ m_c.p = \mathtt{indir}_c[x_c.p]\} \\
\forall p\text{: } 1 \ldots P\text{: } pc_a.p = (pc_c.p \in [c_{10} \ldots c_{18}]\ ?\ a_1 :\ a_2)
\end{array}
$$

where the subscript indicates the concrete or abstract system a variable belongs to, and the remaining variables in Σ_a are identical to the variables occurring in Σ_c.

A. Invariants

We establish some invariants for the concrete system $\mathcal{S}_c$, that will aid us in proving the refinement.

$$
\begin{array}{ll}
I1\text{:} & p \neq q\ \wedge\ pc.p \notin [c_{26} \ldots c_{32}]\ \wedge\ pc.q \notin [c_{26} \ldots c_{32}] \\
& \quad \Rightarrow\ mp.p \neq mp.q \\
I2\text{:} & pc.p \notin [c_{26} \ldots c_{32}]\ \Rightarrow\ \mathtt{indir}[x] \neq mp.p \\
I3\text{:} & x \neq y\ \Rightarrow\ \mathtt{indir}[x] \neq \mathtt{indir}[y]
\end{array}
$$

In the expression of invariants, free variables p and q range over $1 \ldots P$, and x and y range over $1 \ldots N$. Invariants $I1$ and $I2$ indicate that, for any process p, $\mathtt{node}[mp.p]$ can be treated as a "private" node of process p since only process p can modify that. Invariant $I3$ implies that all shared nodes are different. To prove the invariance of *I1* to *I3*, we postulate

$$
\begin{array}{ll}
I4\text{:} & \forall i\text{: } 1 \ldots K\text{: } \mathtt{prot}[i] = \sharp(\{x\text{: } 1 \ldots N \mid \mathtt{indir}[x] = i\}) \\
& \quad +\sharp(\{p \mid (pc_p \notin [c_{26} \ldots c_{32}]\ \wedge\ mp_p = i) \\
& \qquad \vee\ (pc_p = c_{26}\ \wedge\ m_p = i)\}) \\
& \quad +\sharp(\{p \mid pc_p \in [c_{16} \ldots c_{34}]\ \wedge\ pc.p \neq c_{32}\ \wedge\ m_p = i\}) \\
I5\text{:} & pc.p \in [c_{20} \ldots c_{34}]\ \wedge\ pc.p \neq c_{32}\ \wedge\ mp.q = m.p \\
& \quad \Rightarrow\ pc.q \in [c_{26} \ldots c_{32}]
\end{array}
$$

Invariant $I4$ precisely describe the counter $\mathtt{prot}[i]$ for each $i \in 1 \ldots K$. Invariant $I5$ implies that process p cannot read the "private" node of other process q.

Consequently, we have the main reduction theorem for the lock-free implementation using *CAS*:

Theorem 2 *The abstract system $\mathcal{S}_a$ defined in Fig. 1 is implemented by the concrete system $\mathcal{S}_c$ defined in Fig. 2, that is, $\exists \phi : \mathcal{S}_c \sqsubseteq \mathcal{S}_a$.*

For the reason of space, we omit the complete mechanical proof here, and refer the interested reader to [13].

V. Conclusion

The lock-free algorithms often require the use of special atomic processor instructions such as *CAS* or *LL/SC*. However, many machine architectures support either *CAS* or *LL/SC* with restricted semantics. In this paper, we present a Practical lock-free implementation of the ideal semantics of *LL/SC* using only pointer-size *CAS* without causing ABA problem or problems with wrap around. It can be used to provide lock-free functionality for any generic data type. Moreover, to ensure our proof is not flawed, we used the higher-order interactive theorem prover PVS for mechanical support.

Acknowledgment

The Project Sponsored by the Scientific Research Foundation for the Returned Overseas Chinese Scholars, State Education Ministry.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. Theoretical Computer Science, 2(82), 1991.

[2] Z. Manna and A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer Verlag, 1992.

[3] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. 15(5):745-770, November 1993

[4] A. Israeli and L. Rappoport. Disjoint-Access-Parallel implementations of strong shared-memory primitives. In Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing, pages 151-160, August 1994.

[5] J. D. Valois. Implementing lock-free queues. In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, pages 64C69, Las Vegas, NV, 1994.

[6] F. Cassez, C. Jard, B. Rozoy, M. Dermot (Eds.): Modeling and Verification of Parallel Processes. 4th Summer School, MOVEP 2000, Nantes, France, June 19-23, 2000.

[7] M.P. Herlihy and V. Luchangco and M. Moir. The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structure. Proceedings of 16th International Symposium on Distributed Computing, pages 339-353. Springer-Verlag, October 2002.

[8] P. Jayanti and S. Petrovic. Efficient and practical constructionsof ll/sc variables. In Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing, pages 285-294, July 2003.

[9] V. Luchangco and M. Moir and N. Shavit. Nonblocking k-compare-single-swap. Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures, pages 314-323. ACM Press, 2003

[10] H. Gao and W.H. Hesselink. A formal reduction for lock-free parallel algorithms. Proceedings of the 16th Conference on Computer Aided Verification (CAV), July 2004.

[11] S. Doherty, M. Herlihy, V. Luchangco, and M. Moir. Bringing practical lock-free synchronization to 64-bit applications. In Proceedings of the 23rd annual ACM symposium on Principles of distributed computing, pages 31-39, July 2004.

[12] H. Gao, J.F. Groote and W.H. Hesselink. Lock-free Dynamic Hash Tables with Open Addressing. Distributed Computing 17 (2005) 21-42.

[13] W.H. Hesselink. PVS files are available at www.cs.rug.nl/ wim/pub/mans.html.

[14] H. Gao and W.H. Hesselink. A general lock-free algorithm using compare-and-swap. Information and Computation 205 (2007) 225-241.

[15] H. Gao, J.F. Groote and W.H. Hesselink. Lock-free parallel and concurrent garbage collection by mark&sweep. Science of Computer Programming 64 (2007) 341-374.